Continuations

COS 326
Andrew Appel
Princeton University

slides copyright 2018 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
Some Innocuous Code

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0

let big_int = 1000000
let _ = sum big_int
```

What’s going to happen when we run this code?
Some Other Code

Four functions: Green works on big inputs; Red doesn’t.

```plaintext
let rec sum_to (n: int) : int =
    if n > 0 then
        n + sum_to (n-1)
    else 0

let sum_to2 (n: int) : int =
    let rec aux (n:int) (a:int) : int =
        if n > 0 then
            aux (n-1) (a+n)
        else a
    in
    aux n 0

let rec sum (l:int list) : int =
    let rec aux (l:int list) (a:int) : int =
        match l with
        [] -> 0
        | hd::tail -> hd + sum2 tail
    in
    aux l 0

let rec sum2 (l:int list) : int =
    match l with
    [] -> 0
    | hd::tail -> hd + sum2 tail
```

Four functions: Green works on big inputs; Red doesn’t.

```ocaml
define sum_to (n: int) : int =
define aux (n:int) (a:int) : int =
    if n > 0 then
        aux (n-1) (a+n)
    else a

in
aux n 0
```

```ocaml
define sum_to2 (n: int) : int =
define aux (n:int) (a:int) : int =
    if n > 0 then
        aux (n-1) (a+n)
    else a

in
aux n 0
```

```ocaml
define sum (l:int list) : int =
define aux (l:int list) (a:int) : int =
    match l with
    [] -> 0
    | hd::tail -> hd + sum2 tail

in
aux l 0
```

```ocaml
define sum2 (l:int list) : int =
define aux (l:int list) (a:int) : int =
    match l with
    [] -> 0
    | hd::tail -> aux tail (a+hd)

in
aux l 0
```

code that works:

*no computation after recursive function call*
A **tail-recursive function** does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
(* sum of 0..n *)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;
sum big_int;;
```
A **tail-recursive function** does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
sum_to 1000000

--> 1000000 + sum_to 99999
```

```
(* sum of 0..n *)

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;

sum big_int;;
```
A **tail-recursive function** does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```ml
(* sum of 0..n *)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;
let big_int = 1000000;;
sum big_int;;
```

expression size grows at every recursive call ...

lots of adding to do after the call returns"
A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;
let big_int = 1000000;;
sum big_int;;
```
A **tail-recursive function** does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
sum_to 1000000
--> 1000000 + sum_to 99999
--> 1000000 + 99999 + sum_to 99998
--> ...
--> 1000000 + 99999 + 99998 + ... + sum_to 0
--> 1000000 + 99999 + 99998 + ... + 0
```

```
(* sum of 0..n *)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;

let big_int = 1000000;;
sum big_int;;
```

recursion finally bottoms out
Tail Recursion

A *tail-recursive function* does no work after it calls itself recursively.

Not tail-recursive, the substitution model:

```
(* sum of 0..n *)
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else 0
;;
let big_int = 1000000;;
sum big_int;;
```

```
sum_to 1000000
-->
1000000 + sum_to 99999
-->
1000000 + 99999 + sum_to 99998
-->
... 
-->
1000000 + 99999 + 99998 + ... + sum_to 0
-->
1000000 + 99999 + 99998 + ... + 0
-->
... add it all back up ...
```

do a long series of additions to get back an int
Non-tail recursive

let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
Non-tail recursive

```hs
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 10000
Non-tail recursive

```ocaml
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```
Non-tail recursive

```ocaml
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```
Non-tail recursive

```ocaml
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;

sum_to 10000
```
let rec sum_to (n:int) : int =
    if n > 0 then
        n + sum_to (n-1)
    else
        0
;;

sum_to 100
Data Needed on Return Saved on Stack

\[
\text{sum_to 10000} \\
\Rightarrow \\
\ldots \\
\Rightarrow 10000 + 9999 + 9998 + 9997 + \ldots + \\
\Rightarrow \ldots \\
\Rightarrow \ldots \\
\]

not much space left! will run out soon!

every non-tail call puts the data from the calling context on the stack
Memory is partitioned: Stack and Heap

heap space (big!)

stack space (small!)
A **tail-recursive function** is a function that does no work after it calls itself recursively.

Tail-recursive:

```plaintext
sum_to2 1000000

(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```
Tail Recursion

A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```ocaml
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

```
sum_to2 1000000
-->
aux 1000000 0
```
A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```
sum_to2 1000000
```
A **tail-recursive function** is a function that does no work after it calls itself recursively.

Tail-recursive:

```
sum_to2 1000000
--> aux 1000000 0
--> aux 99999 1000000
--> aux 99998 1999999
```

```
(* sum of 0..n *)
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```
A *tail-recursive function* is a function that does no work after it calls itself recursively.

Tail-recursive:

```haskell
let sum_to2 (n: int) :
    int =
    let rec aux (n:int)(a:int) :
        int =
        if n > 0 then
            aux (n-1) (a+n)
        else a
        in
        aux n 0
;;
```

```haskell
sum_to2 1000000
--> aux 1000000 0
--> aux 99999 1000000
--> aux 99998 1999999
--> ...
--> aux 0 (-363189984)
--> -363189984
```

(constant size expression in the substitution model)

(addition overflow occurred at some point)
A **tail-recursive function** is a function that does no work after it calls itself recursively.

```plaintext
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```
A *tail-recursive function* is a function that does no work after it calls itself recursively.

```plaintext
(* sum of 0..n *)
let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

```
stack
```

```
aux 9999 10000
```
A tail-recursive function is a function that does no work after it calls itself recursively.

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```

```
stack

aux 9998 19999
```
A \textit{tail-recursive function} is a function that does no work after it calls itself recursively.

\begin{verbatim}
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;

aux 9997 29998
\end{verbatim}
A *tail-recursive function* is a function that does no work after it calls itself recursively.

```
(* sum of 0..n *)

let sum_to2 (n: int) : int =
  let rec aux (n:int)(a:int) : int =
    if n > 0 then
      aux (n-1) (a+n)
    else a
  in
  aux n 0
;;
```
We used human ingenuity to do the tail-call transform.

Is there a mechanical procedure to transform any recursive function into a tail-recursive one?

not only is sum2 tail-recursive but it reimplements an algorithm that took linear space (on the stack) using an algorithm that executes in constant space!
CONTINUATION-PASSING STYLE CPS!
CPS:

– Short for *Continuation-Passing Style*

– Every function takes a *continuation* (a function) as an argument that expresses "what to do next"

– CPS functions only call other functions as the last thing they do

– All CPS functions are tail-recursive

Goal:

– Find a mechanical way to translate any function in to CPS
Serial Killer or PL Researcher?
Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite in the Adirondacks in 1974.
Confessed to 3 other killings.
Serial Killer or PL Researcher?

Gordon Plotkin
Programming languages researcher
Invented CPS conversion.

Call-by-Name, Call-by Value
and the Lambda Calculus. TCS, 1975.

Robert Garrow
Serial Killer

Killed a teenager at a campsite
in the Adirondacks in 1974.
Confessed to 3 other killings.
Can any non-tail-recursive function be transformed into a tail-recursive one? Yes, if we can capture the *differential* between a tail-recursive function and a non-tail-recursive one.

Idea: Focus on what happens after the recursive call.

```plaintext
let rec sum (l:int list) : int =
  match l with
  []    -> 0
  | hd::tail -> hd + sum tail
;;
```
Can any non-tail-recursive function be transformed into a tail-recursive one? Yes, if we can capture the *differential* between a tail-recursive function and a non-tail-recursive one.

**Idea:** Focus on what happens after the recursive call.

**Extracting that piece:**

```ocaml
let rec sum (l:int list) : int =
  match l with
  | [] -> 0
  | hd::tail -> hd + sum tail

;@
```

**How do we capture it?**
How do we capture that computation?

```
hd + [ ]
```

result of recursive call gets plugged in here

```
fun s -> hd + [ ]
```
Question

How do we capture that computation?

let rec sum (l:int list) : int =
    match l with
    [] -> 0
    | hd::tail -> hd + sum tail
;

fun s -> hd + s

type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
    match l with
    [] -> k 0
    | hd::tail -> sum_cont tail (fun s -> ???) ;;;
How do we capture that computation?

```
let rec sum (l:int list) : int =
  match l with
  | [] -> 0
  | hd::tail -> hd + sum tail

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  | [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s))
```

**Type**

cell

```ocaml
type cont = int -> int;;
```
How do we capture that computation?

```
let rec sum (l:int list) : int =
    match l with
    [] -> 0
    | hd::tail -> hd + sum tail

let rec sum_cont (l:int list) (k:cont): int =
    match l with
    [] -> k 0
    | hd::tail -> sum_cont tail (fun s -> k (hd + s))

let sum (l:int list) : int = ??
```
Question

How do we capture that computation?

```ocaml
let rec sum (l:int list) : int =
    match l with
    | [] -> 0
    | hd::tail -> hd + sum tail

let rec sum_cont (l:int list) (k:cont): int =
    match l with
    | [] -> k 0
    | hd::tail -> sum_cont tail (fun s -> k (hd + s))

let sum (l:int list) : int = sum_cont l (fun s -> s)
```

```ocaml
type cont = int -> int
```

type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
Execution

```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  | [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
-->
  sum_cont [1;2] (fun s -> s)
```
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
    match l with
    [] -> k 0
    | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
--->
    sum_cont [1;2] (fun s -> s)
--->
    sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
type \texttt{cont} = \texttt{int} \to \texttt{int};;

let rec \texttt{sum\_cont} (l: \texttt{int list}) (k:\texttt{cont}) : \texttt{int} =
    match l with
    | [] -> k 0
    | hd::tail -> \texttt{sum\_cont} tail (fun s -> k (hd + s)) ;;

let \texttt{sum} (l: \texttt{int list}) : \texttt{int} = \texttt{sum\_cont} l (fun s -> s)

\texttt{sum [1;2]}
\to
\texttt{sum\_cont [1;2] (fun s -> s)}
\to
\texttt{sum\_cont [2] (fun s -> (fun s -> s) (1 + s));;}
\to
\texttt{sum\_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))}
Execution

type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
-->
  sum_cont [1;2] (fun s -> s)
-->
  sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
  sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (1 + s)) (2 + s)
-->
  (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
    match l with
    [] -> k 0
    | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
-->
sum_cont [1;2] (fun s -> s)
-->
sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->
sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->
(fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
-->
(fun s -> (fun s -> s) (1 + s)) (2 + 0))
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int = 
  match l with
  []   -> k 0
| hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
-->  
  sum_cont [1;2] (fun s -> s)
-->  
  sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
-->  
  sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
-->  
  (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
-->  
  (fun s -> (fun s -> s) (1 + s)) (2 + 0))
-->  
  (fun s -> s) (1 + (2 + 0))
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  [] -> k 0
| hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list) : int = sum_cont l (fun s -> s)

sum [1;2]
--> sum_cont [1;2] (fun s -> s)
--> sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
--> sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
--> (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s)) 0
--> (fun s -> (fun s -> s) (1 + s)) (2 + 0))
--> (fun s -> s) (1 + (2 + 0))
--> 1 + (2 + 0)
--> 3
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s)) ;;

let sum (l:int list): int = sum_cont l (fun s -> s)

sum [1;2]
--> sum_cont [1;2] (fun s -> s)
--> sum_cont [2] (fun s -> (fun s -> s) (1 + s));;
--> sum_cont [] (fun s -> (fun s -> (fun s -> s) (1 + s)) (2 + s))
--> ...
--> 3

Where did the stack space go?
function inside
function inside
function inside
expression

each function is a closure; points to the closure inside it

a stack of closures on the heap

```scheme
sum_cont []
 (fun s3 ->
   (fun s2 ->
     (fun s1 -> s1) (hd1 + s2)
   ) (hd2 + s3)
)
```
function inside function inside function inside expression

a stack of closures on the heap

sum_cont []
(fun s3 ->
 (fun s2 ->
  (fun s1 -> s1) (hd1 + s2)
 ) (hd2 + s3)
)

(function inside function inside function inside expression)

expression

stack

heap
function inside function inside function inside expression

a stack of closures on the heap

```
sum_cont []
  (fun s3 ->
    (fun s2 ->
      (fun s1 -> s1) (hd1 + s2)
    ) (hd2 + s3)
  )
```

```
fun s env ->
  env.k (env.h2 + s)
hd2 = 2
k =

fun s env ->
  env.k (env.hd1 + s)
hd1 = 1
k =

fun s env -> s
```
let rec sum_to (n:int) : int =
    if n > 0 then
        n + sum_to (n-1)
    else
        0
;;

sum_to 100
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 100

but how do you really implement that?
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0

sum_to 100

but how do you really implement that?

there is two bits of information here:
(1) some state (n=100) we had to remember
(2) some code we have to run later
let rec sum_to (n:int) : int =
  if n > 0 then
    n + sum_to (n-1)
  else
    0
;;
sum_to 100

with reality added

code we have to run next

fun s stack ->
  return (stack.n + s)

fun s stack ->
  return (stack.n + s)
n = 100

fun s stack -> return (stack.n+s)

n = 99

fun s stack -> return (stack.n+s)

with the stack

sum_to_cont 98 k3

fun s env -> env.k (env.n + s)

fun s env -> env.k (env.n + s)

with the heap

sum_to_98

return_address

n = 99

return_address

n = 100

with the stack

fun s env -> s

state

sum_to

0x0

98

k3

CPS
Why CPS?

Continuation-passing style is *inevitable*.

It does not matter whether you program in Java or C or OCaml -- there’s code around that tells you “*what to do next*”

- If you explicitly CPS-convert your code, “*what to do next*” is stored on the heap
- If you don’t, it’s stored on the stack

If you take a conventional compilers class, the continuation will be called a *return address* (but you’ll know what it really is!)

The idea of a *continuation* is much more general!
Your compiler can put all the continuations in the heap so you don’t have to (and you don’t run out of stack space)!

Other pros:

- light-weight concurrent threads

Some cons:

- hardware architectures optimized to use a stack
- need tight integration with a good garbage collector

see Empirical and Analytic Study of Stack versus Heap Cost for Languages with Closures. Shao & Appel
Call-backs:

```
request_url : url -> (html -> 'a) -> 'a

request_url "http://www.s.com/i.html" (fun html -> process html)
```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
    match t with
    Leaf -> Leaf
    | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
    ;;

Hint 1: introduce one let expression for each function call:
let x = incr left i in ...

Hint 2: you will need two continuations
CPS Convert the incr function

```ocaml
type tree = Leaf | Node of int * tree * tree

let rec incr (t:tree) (i:int) : tree =
  match t with
    | Leaf -> Leaf
    | Node (j,left,right) -> Node (i+j, incr left i, incr right i)
;;
```

```ocaml
let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
    | Leaf -> k Leaf
    | Node (j,left,right) -> ...
;;
```

```ocaml
type cont = tree -> tree

let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
    | Leaf -> k Leaf
    | Node (j,left,right) -> ...
;;
```
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
    | Leaf -> Leaf
    | Node (j,left,right) -> Node (i+j, incr left i, incr right i) ;;

first continuation: 
Node (i+j, __________, incr right i)

second continuation: 
Node (i+j, left_done, ___________ )
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
  Leaf -> Leaf
  | Node (j,left,right) -> Node (i+j, incr i left, incr i right)
;;

first continuation:  
fun left_done -> Node (i+j, left_done, incr right i)

second continuation:  
fun right_done -> k (Node (i+j, left_done, right_done))
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
    match t with
    Leaf -> Leaf
    | Node (j,left,right) -> Node (i+j, incr left i, incr right i) ;;

fun left_done ->
    let k2 =
        (fun right_done ->
         k (Node (i+j, left_done, right_done))
     )
    in
    incr right i k2
type tree = Leaf | Node of int * tree * tree ;;

let rec incr (t:tree) (i:int) : tree =
  match t with
  | Leaf -> Leaf
  | Node (j, left, right) -> Node (i+j, incr left i, incr right i) ;;

let rec incr_cps (t:tree) (i:int) (k:cont) : tree =
  match t with
  | Leaf -> k Leaf
  | Node (j, left, right) ->
    let k1 = (fun left_done ->
      let k2 = (fun right_done ->
        k (Node (i+j, left_done, right_done)))
      in
      incr_cps right i k2
    )
    in
    incr_cps left i k1 ;;

let incr_tail (t:tree) (i:int) : tree = incr_cps t i (fun t -> t) ;;
FOLD LEFT, FOLD RIGHT
let rec fold_left (f: 'a -> 'b -> 'a) (base: 'a) (bl: 'b list) : 'a =
    match bl with [] -> base | b::bl' -> fold_left f (f base b) bl'

let rec fold_right (f: 'b -> 'a -> 'a) (bl: 'b list) (base: 'a) : 'a =
    match bl with [] -> base | b::bl' -> f b (fold_right f bl' base)

let s1 = fold_left (+) 0 [1;2;3;4;5]
let s2 = fold_right (+) 0 [1;2;3;4;5]
fold_left, fold_right

let rec fold_left (f: 'a -> 'b -> 'a) (base: 'a) (bl: 'b list) : 'a =
  match bl with [] -> base | b::bl' -> fold_left f (f base b) bl’

let rec fold_right (f: 'b -> 'a -> 'a) (bl: 'b list) (base: 'a) : 'a =
  match bl with [] -> base | b::bl' -> f b (fold_right f bl’ base)

let g1 (a: int) (d: int) = a*10+d
let s1 = fold_left g 0 [1;2;3;4;5]

let g2 (d: int) (a: int) = a*10+d
let s2 = fold_right g2 0 [1;2;3;4;5]
CORRECTNESS OF A CPS
TRANSFORM
Are the two functions the same?

```ocaml
type cont = int -> int;;

let rec sum_cont (l:int list) (k:cont): int =
  match l with
  [] -> k 0
  | hd::tail -> sum_cont tail (fun s -> k (hd + s));;

let sum2 (l:int list) : int = sum_cont l (fun s -> s)

let rec sum (l:int list) : int =
  match l with
  [] -> 0
  | hd::tail -> hd + sum tail;;
```

Here, it is really pretty tricky to be sure you've done it right if you don't prove it. Let's try to prove this theorem and see what happens:

```
for all l:int list,
  sum_cont l (fun x -> x) == sum l
```
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
 ... 

case: hd::tail
   IH: sum_cont tail (fun s -> s) == sum tail

   sum_cont (hd::tail) (fun s -> s) ==
for all l:int list, sum_cont l (fun s -> s) == sum l

Proof: By induction on the structure of the list l.

case l = []
  ...

case: hd::tail
  IH: sum_cont tail (fun s -> s) == sum tail

  sum_cont (hd::tail) (fun s -> s)
  == sum_cont tail (fn s' -> (fn s -> s) (hd + s')) (eval)
for all l:int list, sum_cont l (fun s -&gt; s) == sum l

Proof: By induction on the structure of the list l.

case l = []

... 

case: hd::tail
    IH: sum_cont tail (fun s -&gt; s) == sum tail

    sum_cont (hd::tail) (fun s -&gt; s)
    == sum_cont tail (fn s' -&gt; (fn s -&gt; s) (hd + s')) (eval)
    == sum_cont tail (fn s' -&gt; hd + s') (eval)
for all l:int list, \( \text{sum} \_\text{cont} \_l \ (\text{fun} \ s \to s) = = \text{sum} \ l \)

Proof: By induction on the structure of the list \( l \).

case \( l = [] \)
    ...

case: \( \text{hd}::\text{tail} \)
    IH: \( \text{sum} \_\text{cont} \_\text{tail} \ (\text{fun} \ s \to s) = = \text{sum} \ \text{tail} \)

\[
\text{sum} \_\text{cont} \ (\text{hd}::\text{tail}) \ (\text{fun} \ s \to s) \\
= = \text{sum} \_\text{cont} \ \text{tail} \ (\text{fn} \ s' \to (\text{fn} \ s \to s) \ (\text{hd} + s')) \ (\text{eval}) \\
= = \text{sum} \_\text{cont} \ \text{tail} \ (\text{fn} \ s' \to \text{hd} + s') \ (\text{eval})
\]

\[
= = \text{darn!}
\]

we'd like to use the IH, but we can't!
we might like:

\[
\text{sum} \_\text{cont} \ \text{tail} \ (\text{fn} \ s' \to \text{hd} + s') = = \text{sum} \ \text{tail}
\]

... but that's not even true

not the identity continuation (\( \text{fun} \ s \to s \)) like the IH requires
Need to Generalize the Theorem and IH

for all \( l : \text{int list} \),
  \[
  \text{for all } k : \text{int->int}, \quad \text{sum\_cont } l \ k == k \ (\text{sum } l)
  \]
for all l:int list,
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

  must prove: for all k:int->int, sum_cont [] k == k (sum [])
for all `l:int list`,
   for all `k:int->int`, `sum_cont l k == k (sum l)`

Proof: By induction on the structure of the list `l`.

case `l = []`

   must prove: for all `k:int->int`, `sum_cont [] k == k (sum [])`

   pick an arbitrary `k`: 
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

   must prove: for all k:int->int, sum_cont [] k == k (sum [])

   pick an arbitrary k:

   sum_cont [] k
for all l:int list,
    for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

    must prove: for all k:int->int, sum_cont [] k == k (sum [])

    pick an arbitrary k:

        sum_cont [] k
    == match [] with [] -> k 0 | hd::tail -> ... (eval)
    == k 0 (eval)
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

   must prove: for all k:int->int, sum_cont [] k == k (sum [])

pick an arbitrary k:

   sum_cont [] k
== match [] with [] -> k 0 | hd::tail -> ... (eval)
== k 0 (eval)

== k (sum [])
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = []

   must prove: for all k:int->int, sum_cont [] k == k (sum [])

   pick an arbitrary k:

      sum_cont [] k
== match [] with [] -> k 0 | hd::tail -> ... (eval)
== k 0 (eval)
== k (0) (eval, reverse)
== k (match [] with [] -> 0 | hd::tail -> ...) (eval, reverse)
== k (sum [])

case done!
for all l:int list,
   for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

   IH: for all k':int->int, sum_cont tail k' == k' (sum tail)

   Must prove: for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))
for all l:int list,
    for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

    IH:  for all k':int->int, sum_cont tail k' == k' (sum tail)

    Must prove: for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

    Pick an arbitrary k,

        sum_cont (hd::tail) k
for all l:int list,
    for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

    IH: for all k':int->int, sum_cont tail k' == k' (sum tail)

    Must prove: for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

    Pick an arbitrary k,

        sum_cont (hd::tail) k
    == sum_cont tail (fun s -> k (hd + s)) (eval)
for all l:int list,
    for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

    IH: for all k':int->int, sum_cont tail k' == k' (sum tail)

    Must prove: for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

    Pick an arbitrary k,

        sum_cont (hd::tail) k
== sum_cont tail (fun s -> k (hd + s))     (eval)
== (fun s -> k (hd + s)) (sum tail)        (IH with IH quantifier k' replaced with (fun s -> k (hd+s)))
for all l:int list,  
  for all k:int->int, sum_cont l k == k (sum l)

Proof: By induction on the structure of the list l.

case l = [] ===> done!

case l = hd::tail

  IH:  for all k':int->int, sum_cont tail k' == k' (sum tail)

  Must prove:  for all k:int->int, sum_cont (hd::tail) k == k (sum (hd::tail))

  Pick an arbitrary k,

      sum_cont (hd::tail) k
  == sum_cont tail (fun s -> k (hd + s)) (eval)

  == (fun s -> k (hd + s)) (sum tail) (IH with IH quantifier k' replaced with (fun s -> k (hd+s)) (eval, since sum total and and sum tail valuable)

  == k (hd + (sum tail))
Need to Generalize the Theorem and IH

\[
\text{for all } l:\text{int list},
\quad \text{for all } k:\text{int}\rightarrow\text{int}, \quad \text{sum\_cont} \ l \ k = k \ (\text{sum} \ l)
\]

Proof: By induction on the structure of the list \( l \).

\text{case } l = [] \implies \text{done!}

\text{case } l = \text{hd::tail}

IH: \quad \text{for all } k'\!:\text{int}\rightarrow\text{int}, \quad \text{sum\_cont} \ \text{tail} \ k' = k' \ (\text{sum} \ \text{tail})

Must prove: \quad \text{for all } k:\text{int}\rightarrow\text{int}, \quad \text{sum\_cont} \ (\text{hd::tail}) \ k = k \ (\text{sum} \ (\text{hd::tail}))

Pick an arbitrary \( k \),

\[
\begin{align*}
\text{sum\_cont} \ (\text{hd::tail}) \ k \\
\equiv \text{sum\_cont} \ \text{tail} \ (\text{fun} \ s \rightarrow k \ (\text{hd} + s)) \\
\equiv (\text{fun} \ s \rightarrow k \ (\text{hd} + s)) \ (\text{sum} \ \text{tail}) \\
\equiv k \ (\text{hd} + (\text{sum} \ \text{tail})) \\
\equiv k \ (\text{sum} \ (\text{hd::tail}))
\end{align*}
\]

\text{case done!}
QED!
Finishing Up

Ok, now what we have is a proof of this theorem:

\[
\text{for all } l:\text{int list}, \quad \text{for all } k:\text{int->int}, \quad \text{sum\_cont} \ l \ k = k \ (\text{sum} \ l)
\]

We can use that general theorem to get what we really want:

\[
\text{for all } l:\text{int list}, \quad \text{sum2} \ l \\
\quad \equiv \text{sum\_cont} \ l \ (\text{fun } s \rightarrow s) \quad \text{(by eval sum2)} \\
\quad \equiv (\text{fun } s \rightarrow s) \ (\text{sum} \ l) \quad \text{(by theorem, instantiating } k \text{ with } (\text{fun } s \rightarrow s)) \\
\quad \equiv \text{sum} \ l \quad \text{(by eval, since } \text{sum} \ l \text{ valuable)}
\]

So, we've show that the function sum2, which is tail-recursive, is functionally equivalent to the non-tail-recursive function sum.
SUMMARY
CPS is interesting and important:

- **unavoidable**
  - assembly language is continuation-passing

- **theoretical ramifications**
  - fixes evaluation order
  - call-by-value evaluation == call-by-name evaluation

- **efficiency**
  - generic way to create tail-recursive functions
  - Appel's SML/NJ compiler based on this style

- **continuation-based programming**
  - call-backs
  - programming with "what to do next"

- **implementation-technique for concurrency**
We tried to prove the *specific* theorem we wanted:

\[
\text{for all } l : \text{int list}, \text{ sum_cont } l \ (\text{fun } s \rightarrow s) = \text{sum } l
\]

But it didn't work because in the middle of the proof, *the IH didn't apply* -- inside our function we had the wrong kind of continuation -- not \((\text{fun } s \rightarrow s)\) like our IH required. So we had to *prove a more general theorem* about *all* continuations.

\[
\begin{align*}
\text{for all } l : \text{int list}, \\
&\text{for all } k : \text{int} \rightarrow \text{int}, \text{ sum_cont } l \ k = k \ (\text{sum } l)
\end{align*}
\]

This is a common occurrence -- *generalizing the induction hypothesis* -- and it requires human ingenuity. It's why proving theorems is hard. It's also why writing programs is hard -- you have to make the proofs and programs work more generally, around every iteration of a loop.
We developed techniques for reasoning about the space costs of functional programs

- the cost of *manipulating data types* like tuples and trees
- the cost of allocating and using *function closures*
- the cost of *tail-recursive* and non-tail-recursive *functions*

We also talked about some important program transformations:

- *closure conversion* makes nested functions with free variables into pairs of closed code and environment
- the *continuation-passing style* (CPS) transformation turns non-tail-recursive functions in to tail-recursive ones that use no stack space
  - the stack gets moved in to the function closure
- since stack space is often small compared with heap space, it is often necessary to use *continuations and tail recursion*
  - but full CPS-converted programs are unreadable: use judgement